Extreme Scaling of Production Visualization Software on Diverse Architectures

Hank Childs ■ *Lawrence Berkeley National Laboratory*

David Pugmire and Sean Ahern ■ *Oak Ridge National Laboratory*

Brad Whitlock ■ *Lawrence Livermore National Laboratory*

Mark Howison, Prabhat, Gunther H. Weber, and E. Wes Bethel ■ *Lawrence Berkeley National Laboratory*

This article presents the results of experiments studying how the pure-parallelism paradigm scales to massive data sets, including 16,000 or more cores on trillion-cell meshes, the largest data sets published to date in the visualization literature. The findings on scaling characteristics and bottlenecks contribute to understanding how pure parallelism will perform in the future.

Over the last decade, supercomputer capabilities have increased at a staggering rate. Petascale computing has arrived, and machines capable of tens of petaflops will be available in a few years. No end is in sight to this trend, with research in exascale computing well under way. These machines are used primarily for scientific simulations that produce extremely large data sets. The value of these simulations is the scientific insights they produce, which are often enabled by scientific visualization. If visualization software can't keep pace with the massive data sets simulations will produce in the near future, however, it will potentially jeopardize the value of the simulations and thus the supercomputers themselves.

For large-data visualization, the most fundamental question is what paradigm to use to process this data. Most visualization software for large data, including much of the production visualization software that serves large user communities, uses brute-force *pure parallelism*—data parallelism with no optimizations to reduce the amount of data being read. In this approach, the simulation writes data to disk and the visualization software reads this data at full resolution, storing it in primary memory. Because the data is so large, it's necessary to parallelize its processing by partitioning the data over processors and having each processor work on a piece of the problem. Through parallelization, the visualization software can access more I/O bandwidth (to read data faster), more memory (to store more data), and more computing power (to execute its algorithms more quickly).

Our research seeks to better understand how pure parallelism will perform on more cores with larger data sets. How does this technique scale? What are the bottlenecks? What are the pitfalls of running production software at a massive scale? And will pure parallelism be effective for the next generation of data sets?

These questions are especially important because pure parallelism is not the only data-processing paradigm. And where pure parallelism is heavily dependent on I/O bandwidth and large memory footprints, alternatives de-emphasize these traits. Examples include in situ processing, where visualization algorithms operate during the simulation's run, and multiresolution techniques, where a hierarchical version of the data set is created and visualized from coarser to finer versions. With this paper, however, we only study how pure parallelism will handle massive data.

 Published by the IEEE Computer Society

We performed our experiments using only a single visualization tool, VisIt, although we don't believe this limits the impact of our results. We aimed to understand whether pure parallelism will work at extreme scale, not to compare tools. When a program succeeds, it validates the underlying technique. When a program fails, it might indicate a failing in the technique or a poor program implementation. Our principal findings here were that pure parallelism at an extreme scale worked, that algorithms such as contouring and rendering performed well, and that I/O times were very long. Therefore, the only issue requiring further study was I/O performance. We could have addressed this issue by studying other production visualization tools, but they would ultimately employ the same (or similar) low-level I/O calls, such as `fread`, that are themselves the key problem. So, rather than varying visualization tools, each of which follows the same I/O pattern, we varied the I/O patterns (that is, we used collective and noncollective I/O) and compared them across architectures and file systems.

Pure Parallelism

Pure parallelism partitions the underlying mesh (or points for scattered data) of a large data set among its cores, each of which corresponds to a message passing interface (MPI) task. Each core loads its portion of the data set at full resolution, applies visualization algorithms to its piece, and then combines the results, typically through rendering. In VisIt, the pure-parallelism implementation centers around data-flow networks. To satisfy a given request, every core sets up an identical data-flow network, differentiated only by the portion of the whole data set on which that core operates. (For previous work on this area, see the "Related Work in Large-Data Visualization" sidebar.)

Many visualization algorithms are *embarassingly parallel*; that is, they require no interprocess communication and can operate on their own portion of the data set without coordination with the other cores. Examples of these algorithms are slicing and contouring. However, some important algorithms do require interprocess communication and therefore aren't embarassingly parallel. Examples include volume rendering, streamline generation, and ghost data generation. (When a large data set is decomposed into chunks, ghost data is a redundant layer of cells around the boundaries of each chunk. These extra cells are sometimes necessary to prevent artifacts, usually due to interpolation inconsistencies.)

The pure-parallelism paradigm accommodates both types of algorithms. For embarrassingly parallel algorithms, each core can directly apply the serial algorithms to its portion of the data set. Pure parallelism is often the simplest environment to implement non-embarrassingly parallel algorithms as well, because every piece of data is available at any time, at full resolution. This property is especially beneficial when the operation order is data dependent (streamlines) or when coordination between the data chunks is necessary (volume rendering, ghost data generation).

After the algorithms are applied, their results are rendered in parallel. The rendering algorithm combines all the cores' results, as if all the data was rendered on a single core. The algorithm scales relatively well, although the combination phase is $O(n \log n)$.

Pure parallelism typically employs one of two hardware scenarios. Processing occurs

- on a smaller supercomputer dedicated to visualizing and analyzing data sets produced by a larger supercomputer or
- on the supercomputer that generated the data.

In both scenarios, visualization and analysis programs often operate with substantially less resources than the simulation code for the same data set. For either hardware scenario, the rule of thumb for pure parallelism is to have approximately 10 percent of the total memory footprint used to generate the data. Although rising hardware costs have relaxed this rule somewhat for the largest supercomputers, many US supercomputing centers are procuring dedicated machines that come close to this guideline. For example, Lawrence Livermore National Laboratory's Gauss machine has 8 percent of the memory of the Blue Gene/L machine, and Argonne National Laboratory's Eureka has nearly 5 percent of the memory of the Intrepid machine. Our research for this article was done with the second scenario, on the supercomputer itself, but our results apply to either hardware scenario.

Massive-Data Experiments

Our basic experiment used a parallel program with high concurrency to read in a very large data set, apply a contouring algorithm (Marching Cubes), and render this surface as a 1,024 × 1,024 image. We originally wanted to also perform volume rendering but encountered difficulties (which we describe later). An unfortunate reality of experiments of this nature is that running large jobs on the largest supercomputers in the world is a difficult and opportunistic undertaking. After improving our volume-rendering algorithm, we couldn't rerun our experiments on all these machines with

Related Work in Large-Data Visualization

Alternatives to pure parallelism include in situ processing,[1,2] multiresolution processing,[3,4] out-of-core processing,[5] and data subsetting.[6,7] Framing the decision of which paradigm to use to process massive data as a competition between pure parallelism and the others is an oversimplification. These techniques have various strengths and weaknesses and are often complementary. From our perspective, the issue is whether pure parallelism will scale sufficiently to process massive data sets.

Our study employed the VisIt visualization tool,[8] which primarily uses pure parallelism, although some of its algorithms allow for out-of-core processing, data subsetting, or in situ processing. (The experiments in the main article used pure parallelism exclusively.) ParaView,[9] another viable choice for our study, also relies heavily on pure parallelism, again with options for out-of-core processing, data subsetting, and in situ visualization. These tools' end users, however, use pure parallelism almost exclusively, using the other paradigms only situationally. Both tools rely on the Visualization ToolKit (VTK),[10] which provides relatively small memory overhead for large data sets. This was crucial for our study (because data sets must fit in memory) and especially important given the trend in petascale computing toward low-memory machines.

The parallel VTK/ParaView infrastructure, in the context of this pure-parallelism article, is highly similar to the VisIt implementation in that they both divide the data set into pieces, partition those pieces, operate in an embarrassingly parallel fashion when possible, and perform parallel rendering. So, we believe our scalability results are applicable to the major open-source large data visualization tools in use today. Yet another viable choice to explore pure parallelism would have been the commercial product EnSight,[11] but accurately measuring performance with it would have been more difficult.

We believe this effort is the first to examine the performance of pure parallelism at extreme scale on diverse architectures. However, other publications provide corroboration in this space, albeit as individual data points. For example, Tom Peterka and his colleagues demonstrated a similar overall balance of I/O and computation time when volume-rendering a 90-billion-cell data set on a Blue Gene/P machine.[12]

References

1. C.R. Johnson, S. Parker, and D. Weinstein, "Large-Scale Computational Science Applications Using the SCIRun Problem Solving Environment," *Proc. Int'l Supercomputing Conf.* (ISC 00), 2000; http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.140.2320&rep=rep1&type=pdf.
2. R. Haimes, "pV3: A Distributed System for Large-Scale Unsteady CFD Visualization," paper 94-0321, Am. Inst. Aeronautics and Astronautics, 1994.
3. J. Clyne et al., "Interactive Desktop Analysis of High Resolution Simulations: Application to Turbulent Plume Dynamics and Current Sheet Formation," *New J. Physics*, Aug. 2007, p. 301.
4. V. Pascucci and R.J. Frank, "Global Static Indexing for Real-Time Exploration of Very Large Regular Grids," *Proc. 2001 ACM/IEEE Conf. Supercomputing* (SC 01), IEEE CS Press, 2001, p. 2.
5. C. Silva et al., "Out-of-Core Algorithms for Scientific Visualization and Computer Graphics," *Course Notes from Proc. IEEE Visualization*, 2002; http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.15.6463.
6. H. Childs et al., "A Contract Based System for Large Data Visualization," *Proc. 16th IEEE Conf. Visualization* (VIS 05), IEEE CS Press, 2005, p. 25.
7. O. Rübel et al., "High Performance Multivariate Visual Data Exploration for Extremely Large Data," *Proc. 2008 ACM/IEEE Conf. Supercomputing* (SC 08), IEEE CS Press, 2008, pp. 1–12.
8. H. Childs and M. Miller, "Beyond Meat Grinders: An Analysis Framework Addressing the Scale and Complexity of Large Data Sets," *Proc. 2006 High-Performance Computing Symp.* (HPC 06), Soc. for Modeling and Simulation Int'l (SCS), 2006, pp. 181–186.
9. C.C. Law, A. Henderson, and J. Ahrens, "An Application Architecture for Large Data Visualization: A Case Study," *Proc. IEEE 2001 Symp. Parallel and Large-Data Visualization and Graphics* (PVG 01), IEEE Press, 2001, pp. 125–128.
10. W.J. Schroeder, K.M. Martin, and W.E. Lorensen. "The Design and Implementation of an Object-Oriented Toolkit for 3D Graphics and Visualization," *Proc. 7th IEEE Conf. Visualization* (VIS 96), IEEE CS Press, 1996, pp. 93–ff.
11. *EnSight User Manual*, ver. 9.0, Computational Eng. Int'l, 2008; www2.ensight.com/90_manuals/90-UserManual.pdf.
12. T. Peterka et al., "End-to-End Study of Parallel Volume Rendering on the IBM Blue Gene/P," *Proc. 2009 Int'l Conf. Parallel Processing* (ICPP 09), IEEE CS Press, 2009, pp. 566–573.

the improved volume-rendering code. Furthermore, real-world issues such as I/O and network contention undoubtedly affected the performance of these runs. Although we only studied isosurfacing, the process of loading data, applying an algorithm, and rendering is representative of many visualization operations, and involves a significant portion of the code base.

Our variations of this experiment fell into three categories. The first was diverse supercomputing environments. We tested these techniques' viability with different operating systems, I/O behavior, computing power (flops), and network characteristics.

We performed these tests on

- two Cray XT machines (Oak Ridge National Laboratory's JaguarPF and Lawrence Berkeley National Laboratory's Franklin),

Table 1. Characteristics of the supercomputers in this study.

Machine name	Machine type or OS	Total no. of cores	Memory per core (Gbytes)	System type	Clock speed	Peak flops	Top 500 rank (as of Nov. 2009)
JaguarPF	Cray	224,162	2.0	XT5	2.6 GHz	2.33 Pflops	1
Ranger	Sun Linux	62,976	2.0	Opteron Quad	2.0 GHz	503.8 Tflops	9
Dawn	Blue Gene/P	147,456	1.0	PowerPC	850.0 MHz	415.7 Tflops	11
Franklin	Cray	38,128	1.0	XT4	2.6 GHz	352 Tflops	15
Juno	Commodity (Linux)	18,402	2.0	Opteron Quad	2.2 GHz	131.6 Tflops	27
Purple	AIX (*A*dvanced *I*nteractive *E*xecutive)	12,208	3.5	Power5	1.9 GHz	92.8 Tflops	66

- a Sun Linux machine (the Texas Advanced Computing Center's Ranger),
- a Chaos Linux machine (Lawrence Livermore National Laboratory's Juno),
- an AIX (*A*dvanced *I*nteractive *E*xecutive) machine (Lawrence Livermore's Purple), and
- a Blue Gene/P machine (Lawrence Livermore's Dawn).

Table 1 provides details about these machines. For all but Purple, we ran with 16,000 cores and visualized one trillion cells. (On Purple, we ran with only 8,000 cores and a half trillion cells because the full machine has only 12,208 cores and only 8,000 are easily obtainable for large jobs.) For JaguarPF and Franklin, which had more than 16,000 cores available, we performed a weak-scaling study, maintaining a ratio of one trillion cells for every 16,000 cores (see Figure 1).

The second category was I/O pattern. We tested whether certain patterns (collective versus noncollective) exhibit better performance at scale. For the noncollective tests, we generated the data as compressed binary data (gzipped). We used 10 files for every core; every file contained 6.25 million data points, for a total of 62.5 million data points per core. Because simulation codes often write out one file per core and, as a rule of thumb, visualization codes receive one-tenth of the cores of the simulation code, we used multiple files per core to best emulate common real-world conditions. Because this pattern might not be optimal for I/O access, we performed a separate test in which all cores used collective access on a single, large file via MPI-IO.

The third category was data generation. Our primary mechanism was to upsample data by interpolating a scalar field for a smaller mesh onto a high-resolution rectilinear mesh. However, to offset concerns that upsampled data might be unrepresentatively smooth, we ran a second experiment, in which the large data set replicated a small data set many times over. The source data set was a core-collapse supernova simulation from the Chimera code on a curvilinear mesh of more than 3.5 million cells. (The sample data was courtesy of Tony Mezzacappa and Bronson Messer from Oak Ridge, Steve Bruenn from Florida Atlantic University, and Reuben Budjiara from the University of Tennessee.) We applied these upsampling and replication approaches because we aren't aware of any data sets containing trillions of cells. Moreover, our study's primary objective was to better understand the performance and functional limits of parallel visualization software, which can be achieved using synthetic data.

Figure 1. Our first category of experiments varied over supercomputing environment. This image is from the Franklin run, showing a contour of a 32,000-core VisIt visualization of a two-trillion-cell data set.

Varying over the Supercomputing Environment

We ran these experiments on different supercomputers and kept the I/O pattern and data generation fixed, using noncollective I/O and upsampled data generation. Figure 2 and Table 2 show the results.

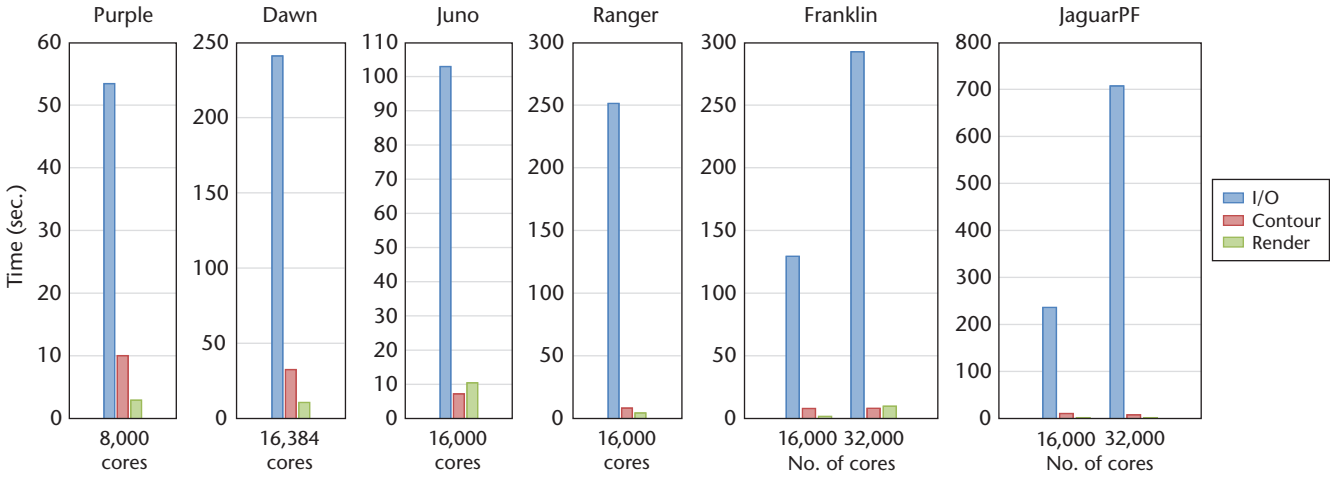


Figure 2. Runtimes for I/O, contouring, and rendering. These results show that, although there is variation across the supercomputers, I/O is the slowest phase.

Table 2. Performance across diverse architectures.

Machine	No. of cores	Data set size (TCells)	Total I/O time (sec.)	Contour time (sec.)	Total pipeline execution time (sec.)†	Rendering time (sec.)
Purple	8,000	0.5	53.4	10.0	63.7	2.9
Dawn	16,384*	1.0	240.9	32.4	277.6	10.6
Juno	16,000	1.0	102.9	7.2	110.4	10.4
Ranger	16,000	1.0	251.2	8.3	259.7	4.4
Franklin	16,000	1.0	129.3	7.9	137.3	1.6
JaguarPF	16,000	1.0	236.1	10.4	246.7	1.5
Franklin	32,000	2.0	292.4	8.0	300.6	9.7
JaguarPF	32,000	2.0	707.2	7.7	715.2	1.5

** Dawn requires that the number of cores be a power of two.*
† This measure indicates the time to produce the surface.

Four observations are noteworthy. First, careful consideration of I/O striping parameters is necessary for optimal I/O performance on Lustre file systems (Franklin, JaguarPF, Ranger, Juno, and Dawn). Even though JaguarPF has more I/O resources than Franklin, its I/O performance was worse because its default stripe count is four. In contrast, Franklin's default stripe count of two was better suited for the I/O pattern, which read 10 separate gzipped files per core. Smaller stripe counts often benefit file-per-core I/O because the files are usually small enough (tens of megabytes) that they won't contain many stripes. Spreading them thinly over many I/O servers increases contention.

Second, because the data was gzipped, the I/O load across cores was unequal. The reported I/O times measure the elapsed time between file opening and a barrier after all cores are finished reading. Because of this load imbalance, I/O time didn't scale linearly from 16,000 to 32,000 cores on Franklin and JaguarPF.

Third, Dawn has the slowest clock speed (850 MHz), which was reflected in its contouring and rendering times.

Finally, although many of the variations we observed were expected—for example, owing to slow clock speeds, interconnects, or I/O servers—others weren't. When we increased Franklin's rendering time from 16,000 to 32,000 cores, seven to 10 network links failed and had to be statically rerouted, resulting in suboptimal network performance. Rendering algorithms are "all reduce"-type operations sensitive to bisectional bandwidth, which was affected by this issue. Also, for Juno's slow rendering time, we suspect a similar network problem. We haven't been able to schedule time on either machine to follow up on these issues.

Varying over the I/O Pattern

We compared collective and noncollective I/O patterns on Franklin for a one-trillion-cell upsampled data set. In the noncollective test, each core performed 10 pairs of `fopen` and `fread` calls on independent gzipped files without any coordination among cores. In the collective test, all cores synchronously called `MPI_File_open` once and then `MPI_File_read_at_all` 10 times on a shared file (each read call corresponded to a differ-

Table 3. The performance of different I/O patterns on Franklin.

I/O pattern	No. of cores	Data set size (TCells)	Total I/O time (sec.)	Data read (Gbytes)	Read bandwidth (Gbytes per second)
Collective	16,016	1	478.3	3,725.3	7.8
Noncollective	16,000	1	129.3	954.2	7.4

ent domain in the data set). An underlying collective buffering, or two-phase algorithm, in Cray's MPI-IO implementation aggregated read requests onto a subset of 48 nodes (matching the file's 48 stripe count) that coordinated the low-level I/O workload, dividing it into 4-Mbyte stripe-aligned `fread` calls. As the 48 aggregator nodes filled their read buffers, they shipped the data through MPI to its final destination among the 16,016 cores. We used a different number of cores (16,000 versus 16,016) to make data layout more convenient for each scheme.

Table 3 shows the I/O patterns' performance on Franklin. The data set size for collective I/O corresponds to 4 bytes for one trillion cells. The data read isn't 4,000 Gbytes because 1 Gbyte is 1,073,741,824 bytes. The data set size for noncollective I/O is much smaller because it was gzipped.

Both patterns led to similar read bandwidths, 7.4 and 7.8 Gbytes per second (GBps), which are about 60 percent of the maximum available bandwidth of 12 GBps on Franklin. In the noncollective case, load imbalances caused by different gzip compression factors might account for this discrepancy. For the collective I/O, we speculate that coordination overhead between the MPI tasks might limit efficiency. Furthermore, achieving 100 percent efficiency wouldn't substantially change the balance between I/O and computation.

Varying over Data Generation

Here, we processed both upsampled and replicated data sets with one trillion cells on 16,016 cores of Franklin using collective I/O. Figure 3 shows the visualization results for the replicated data set.

The contouring times were identical because this operation is dominated by the movement of data through the memory hierarchy (L2 cache to L1 cache to registers), rather than the relatively rare case in which a cell contains a contribution to the isosurface (see Table 4). The rendering time nearly doubled because the contouring algorithm produced more triangles with the replicated data set.

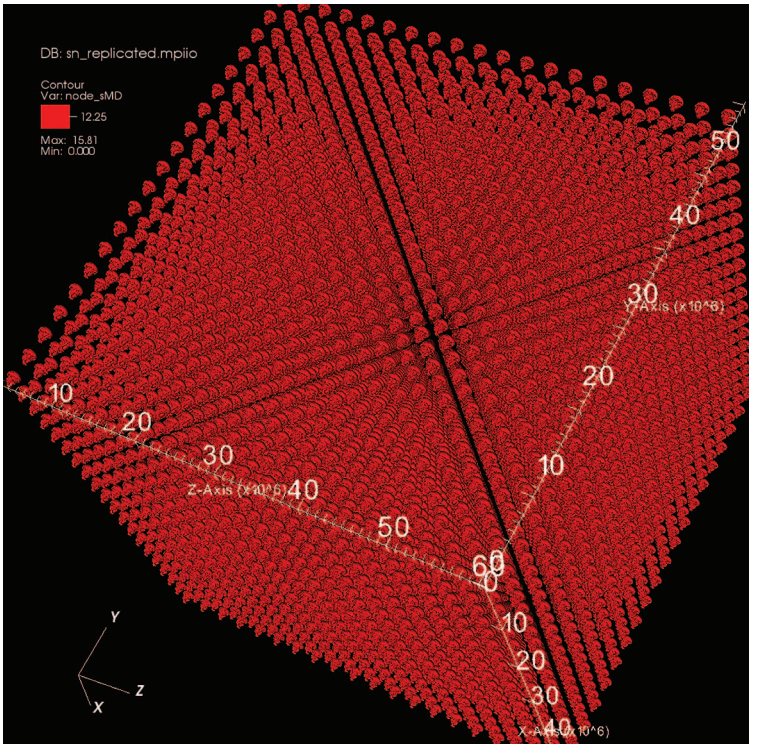


Figure 3. Our third category of experiments varied over data generation, to ensure we weren't studying data that was unrepresentatively smooth. This image shows a contouring of replicated data (one trillion cells total), visualized with VisIt on Franklin using 16,016 cores.

Table 4. Performance across different data generation methods.

Data generation	Total I/O time (sec.)	Contour time (sec.)	Total pipeline execution time (sec.)	Rendering time (sec.)
Upsampled	478.3	7.6	486.0	2.8
Replicated	493.0	7.6	500.7	4.9

Scaling Experiments

To further demonstrate the scaling properties of pure parallelism, we present results that demonstrate weak scaling (scaling up the number of processors with a fixed amount of data per processor) for both isosurface generation and volume rendering. (We ran this study in July 2009, after fixing the volume-rendering algorithm.) Once again, these algorithms test a large portion of the underlying pure-parallelism infrastructure and indicate a strong likelihood of weak scaling for other algorithms in this setting. Demonstrating weak-scaling properties on high-performance computing systems meets the accepted standards of Joule certification, which is a US Office of Management and Budget program to evaluate the effectiveness of agency programs, policies, and procedures.

Study Overview

We performed the scaling studies on output from Denovo, Oak Ridge National Laboratory's 3D

Table 5. Weak scaling of isosurfacing.

Algorithm	No. of cores	Time (sec.)		
		Minimum	Maximum	Average
Calculate radiation*	4,096	0.180	0.250	0.2100
	12,270	0.190	0.250	0.2200
Isosurface†	4,096	0.014	0.027	0.0180
	12,270	0.014	0.027	0.0170
Render (on core)‡	4,096	0.020	0.065	0.0225
	12,270	0.021	0.069	0.0230
Render (across cores)‖	4,096	0.048	0.087	0.0520
	12,270	0.050	0.091	0.0530

* *The time to calculate the linear combination of the 27 scalar fluxes.*
† *The isosurface algorithm's execution time.*
‡ *The time to render that core's surface.*
‖ *The time to combine that image with the other cores' images.*

Table 6. Weak scaling of volume rendering.

No. of cores	Processing time per ray (sec.)		
	1,000 samples	2,000 samples	4,000 samples
4,096	7.21	4.56	7.54
12,270	6.53	6.60	6.85

radiation transport code that models radiation dose levels for a nuclear reactor core and its surrounding areas. The Denovo simulation code doesn't directly output a scalar field representing the effective dose. Instead, we calculated this dose at runtime through a linear combination of 27 scalar fluxes. For both the isosurface and volume-rendering tests, VisIt read in 27 scalar fluxes and combined them to form a single scalar field representing radiation dose levels. The isosurface extraction test extracted six evenly spaced isocontour values of the radiation dose levels and rendered a 1,024 × 1,024 pixel image. The volume-rendering test consisted of ray casting with 1,000, 2,000 and 4,000 samples per ray of the radiation dose level on a 1,024 × 1,024 pixel image.

We ran these visualization algorithms on a baseline Denovo simulation consisting of 103,716,288 cells on 4,096 spatial domains with a 83.5-Gbyte disk. We ran the second test on a Denovo simulation nearly three times the size of the baseline run, with 321,117,360 zones on 12,720 spatial domains and a 258.4-Gbyte disk.

Results

The baseline calculation used 4,096 cores; the larger calculation used 12,270. We chose these core counts, which are large relative to the problem size, because they represent the number of cores Denovo used. This matching core count was important for the Joule study and is also indicative of performance for an in situ approach.

Tables 5 and 6 give the weak-scaling results of isosurfacing and volume rendering, respectively. (These tests didn't include I/O.) The algorithm demonstrates superlinear performance because the number of samples per core (which directly affects the work performed) is smaller at 12,270 cores, whereas the number of cells per core is constant. The anomaly in which performance increases at 2,000 samples per ray requires further study.

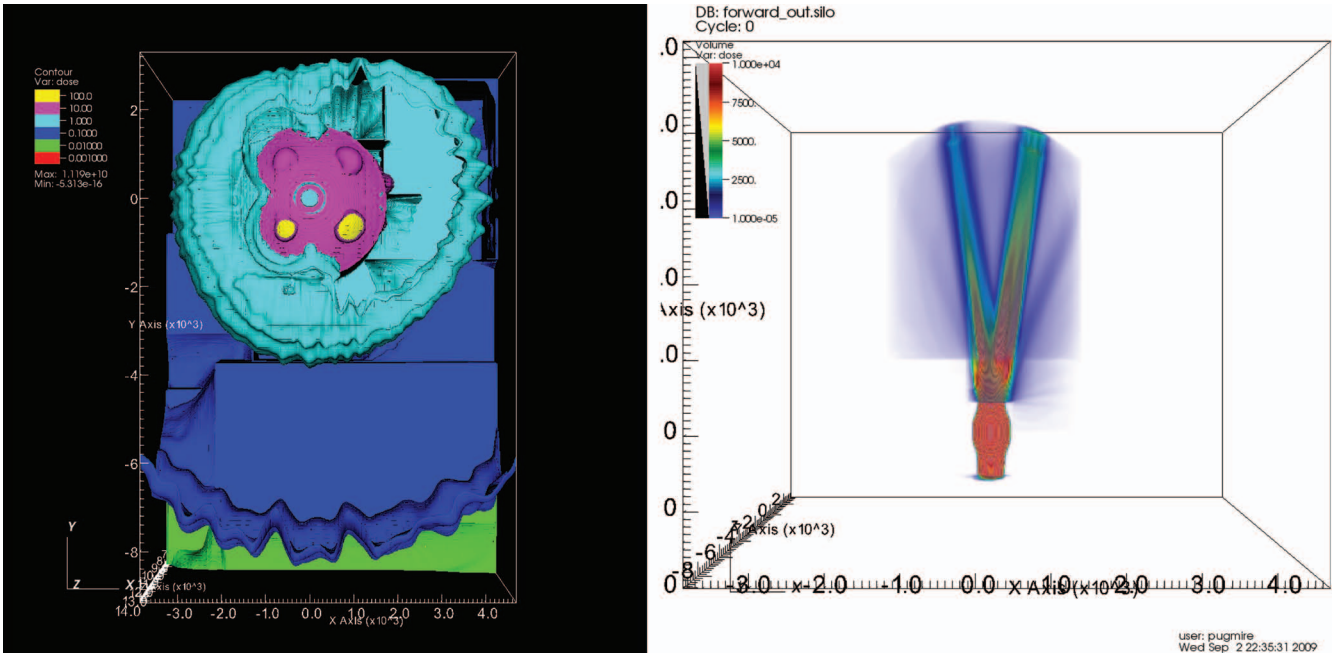
Figure 4a shows the rendering of an isosurface from the Denovo calculation we produced using VisIt; Figure 4b gives the volume rendering of the data from the calculation.

Pitfalls at Scale

Our results in this section illustrate that decisions that were appropriate on the order of hundreds of cores become serious impediments at higher levels of concurrency. The offending code existed at various levels of the software, from core algorithms (volume rendering), to code supporting the algorithms (status updates), to foundational code (plug-in loading). The volume-rendering and status update problems were easily correctable; their fixes will be in the next public version of VisIt. We partially addressed the plug-in loading problem, but a total fix might require removing shared libraries altogether.

Volume Rendering

The volume-rendering code used an $O(n^2)$ buffer, where n is the number of cores. An all-to-all communication phase redistributed samples along rays according to a partition with dynamic assignments. An optimization for this phase minimized the number of samples that needed to be communicated by favoring assignments that kept samples on their originating core. This optimization required an $O(n^2)$ buffer that contained mostly zeroes. Although this was effective for small core counts, the coordination overhead caused VisIt to



(a) (b)

Figure 4. Visualization results for the Denovo calculation, produced by VisIt using 12,270 cores of JaguarPF: (a) a rendering of an isosurface and (b) a volume rendering of the data. These images were the byproduct of the tests to demonstrate that VisIt and the pure-parallelism technique that it uses are capable of weak scaling.

run out of memory at scale. Our solution was to eschew the optimization, simply assigning pixels to cores without concern for where individual samples lay. As the number of samples decreases with large core counts, ignoring this optimization altogether at high concurrency is probably the best course of action.

We don't have comprehensive volume-rendering data to present for the one-trillion-cell data sets. However, we observed that after our changes, ray-casting performance was approximately five seconds per frame for a 1,024 × 1,024 image (see Figure 5).

For the weak-scaling study on Denovo data, running with 4,096 cores, the speedup was approximately a factor of five (see Table 7).

Table 7. Volume rendering of Denovo data at 4,096 cores before and after speedup.

	Processing time per ray (sec.)		
Date run	**1,000 samples**	**2,000 samples**	**4,000 samples**
Spring 2009	34.70	29.00	31.50
Summer 2009	7.21	4.56	7.54

All-to-One Communication

At the end of every pipeline execution, each core reports its status (success or failure) and some metadata (such as extents). These status and extents were being communicated from each MPI task to MPI task 0 through point-to-point communication, which caused significant delays, as Table 8 shows.

After our first round of experiments, our colleague Mark Miller of Lawrence Livermore Lab independently observed the same problem and reimplemented the scheme to use tree communication. Taking the pipeline time and subtracting contour and I/O time approximates how much time was spent waiting for status and extents updates. (The other runs reported in this article had status-checking code disabled; the last Dawn run is the only reported run with new status code.)

Another pitfall is the difficulty in getting consistent results. In the Dawn runs, a dramatic slowdown in I/O times occurred from June to August. This is because the I/O servers backing the file system became unbalanced in their disk usage in

Table 8. Performance with old versus new status-checking code, on Dawn.

All-to-one?	No. of cores	Data set size (TCells)	Total I/O time (sec.)	Contour time (sec.)	Total pipeline execution time (sec.)	Pipeline minus contour & I/O (sec.)	Date run
Yes	16,384	1	88.0	32.2	368.7	248.5	June 2009
Yes	65,536	4	95.3	38.6	425.9	294.0	June 2009
No	16,384	1	240.9	32.4	277.6	4.3	Aug. 2009

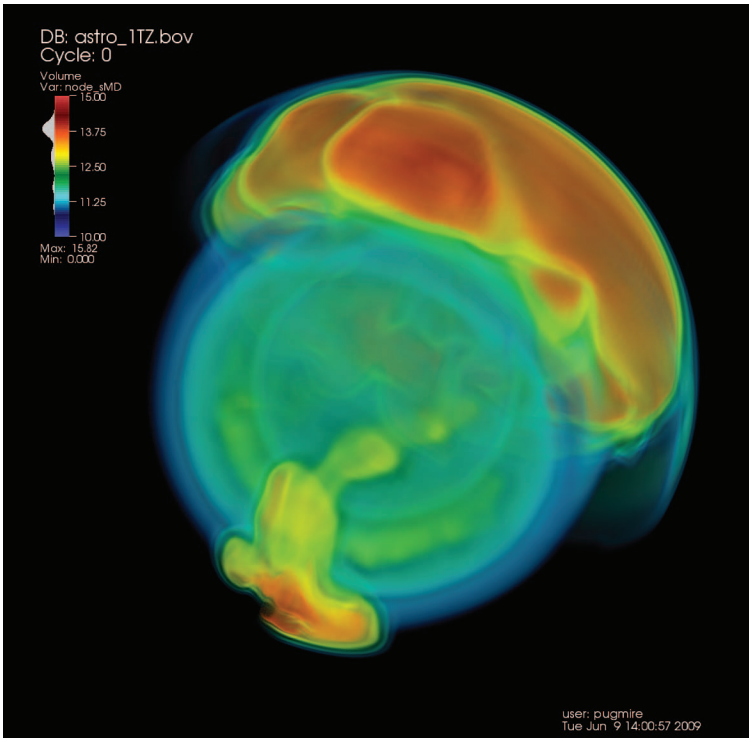


Figure 5. Volume rendering of one trillion cells, visualized by VisIt on JaguarPF. As expected, we ran into many pitfalls when running at high levels of concurrency. In this case, VisIt's volume-rendering algorithm had to be modified to remove an $O(n^2)$ algorithm.

Our results demonstrate that pure parallelism does scale but is only as good as its supporting I/O infrastructure.

July. This caused the algorithm that assigns files to servers to switch from a round-robin scheme to a statistical scheme, meaning files were no longer assigned uniformly across I/O servers. Although this scheme makes sense from an operating system perspective by leveling out the storage imbalance, it hampers access times for end users. With the new scheme, the number of files assigned to each I/O server followed a Poisson distribution, with some servers assigned three or four more times more files than others. Because each I/O server has a fixed bandwidth, those with more files will take longer to serve up data, resulting in I/O performance degradation of factors of three or four for the cores trying to fetch data from the overloaded I/O servers.

Shared Libraries and Start-Up Time

During our first runs on Dawn, using only 4,096 cores, we observed lags in start-up time that worsened as the core count increased. Each core was reading plug-in information from the file system, creating contention for I/O resources. We addressed this problem by modifying VisIt's plug-in infrastructure so that plug-in information could be loaded on MPI task 0 and broadcast to other cores. This change made plug-in loading nine times faster.

That said, start-up time was still slow, taking as long as five minutes. VisIt uses shared libraries in many instances to let new plug-ins access symbols not used by current VisIt routines; compiling statically would remove these symbols. The likely path forward is to compile static versions of VisIt for the high-concurrency case. This approach will likely be palatable because new plug-ins are frequently developed at lower levels of concurrency.

Our results demonstrate that pure parallelism does scale but is only as good as its supporting I/O infrastructure. We successfully visualized up to four trillion cells on diverse architectures with production visualization software. The supercomputers we used were "underpowered," in that the current simulation codes on these machines produce meshes far smaller than a trillion cells. They were appropriately sized, however, when considering the rule of thumb that the visualization task should get 10 percent of the simulation task's resources and assuming our trillion-cell mesh represents the simulation of a hypothetical 160,000-core machine.

I/O performance became a major focus of our study because slow I/O prevented interactive rates when loading data. Most supercomputers are configured for I/O bandwidth to scale with the number of cores, so the bandwidths we observed in our experiments are commensurate with what we should expect when using 10 percent of a future supercomputer. Thus, the inability to read data sets quickly presents a real concern. Worse, the latest supercomputing trends show diminishing I/O relative to increasing memory and flops, meaning that the I/O bottleneck we observed might potentially constrict further with the next generation of supercomputers.

Some potential hardware and software solutions might help address this problem, however. From the software side, multiresolution techniques and data subsetting (such as query-driven visualization) limit how much data is read, whereas in situ visualization avoids I/O altogether. From the hardware side, an increased focus on balanced machines that have I/O bandwidth commensurate with computing power would reduce I/O time. Furthermore, emerging I/O technologies, such as flash drives, might have a significant impact. From this study,

we conclude that some combination of these solutions will be necessary to overcome the I/O problem and obtain good performance.

Acknowledgments

This work was supported by the Director, Office of Advanced Scientific Computing Research, Office of Science, of the US Department of Energy (DOE) under contract DE-AC02-05CH11231 through the Scientific Discovery through Advanced Computing program's Visualization and Analytics Center for Enabling Technologies. We thank Mark Miller for status update improvements and the anonymous reviewers, whose suggestions greatly improved this article. The following resources contributed to our research results: the National Energy Research Scientific Computing Center (NERSC), which is supported by the US DOE Office of Science under contract DE-AC02-05CH11231; the Livermore Computing Center at Lawrence Livermore National Laboratory (LLNL), which is supported by the US DOE National Nuclear Security Administration under contract DE-AC52-07NA27344; the Center for Computational Sciences at Oak Ridge National Laboratory (ORNL), which is supported by the US DOE Office of Science under contract De-AC05-00OR22725; and the Texas Advanced Computing Center (TACC) at the University of Texas at Austin, which provided HPC resources. We thank the personnel at the computing centers that helped us perform our runs, specifically Katie Antypas, Kathy Yelick, Francesca Verdier, and Howard Walter of NERSC; Paul Navratil, Kelly Gaither, and Karl Schulz of TACC; James Hack, Doug Kothe, Arthur Bland, and Ricky Kendall of ORNL's Leadership Computing Facility; and David Fox, Debbie Santa Maria, and Brian Carnes of LLNL's Livermore Computing.

Hank Childs *is a computer systems engineer at Lawrence Berkeley National Laboratory and a researcher at the University of California, Davis. His research interests include parallel visualization and production visualization applications. Childs has a PhD in computer science from the University of California at Davis. Contact him at hchilds@lbl.gov.*

David Pugmire *is a computer scientist in Oak Ridge National Laboratory's Scientific Computing Group. His research interest is parallel scientific data analysis and visualization. Pugmire has a PhD in computer science from the University of Utah. Contact him at pugmire@ornl.gov.*

Sean Ahern *is the visualization task leader for the Oak Ridge Leadership Computing Facility at Oak Ridge National Laboratory and the director of the US National Science Foundation TeraGrid XD Center for Remote Data Analysis and Visualization at the University of Tennessee. His research interests are distributed visualization and data processing on computational clusters. Ahern has a BS in computer science and mathematics from Purdue University. Contact him at ahern@ornl.gov.*

Brad Whitlock *is a computer scientist at Lawrence Livermore National Laboratory and a founding developer of the VisIt visualization and data-analysis software. His interests include visualization, parallel programming, and GUI design. Whitlock has a BS in computer science from California State University, Sacramento. Contact him at whitlock2@llnl.gov.*

Mark Howison *is a computer systems engineer in Lawrence Berkeley National Laboratory's Visualization Group. His research interests include scientific computing, visualization, graphics, and parallel I/O. Howison has an MS in computer science from the University of California, Berkeley. Contact him at mhowison@lbl.gov.*

Prabhat *is a member of Lawrence Berkeley National Laboratory's Scientific Visualization Group. His research interests include computer graphics, scientific visualization, high-performance rendering and computing, human-computer interaction, information visualization, general-purpose computation on GPUs, and machine learning. Prabhat has an MS in computer science from Brown University. He's a member of the ACM. Contact him at prabhat@lbl.gov.*

Gunther H. Weber *is a research scientist and engineer at Lawrence Berkeley National Laboratory and an adjunct assistant professor at the University of California, Davis. His research interests include topology-based data analysis and parallel visualization. Gunther has a PhD in computer science from the University of Kaiserslautern. He's a member of the ACM and the IEEE Computer Society. Contact him at ghweber@lbl.gov.*

E. Wes Bethel *is a staff scientist at Lawrence Berkeley National Laboratory, where he conducts and leads research, development, and deployment activities in high-performance, parallel visual data exploration algorithms and architectures. Bethel has a PhD in computer science from the University of California, Davis. He's a member of ACM Siggraph and IEEE. Contact him at ewbethel@lbl.gov.*